On the Design and Development of Program Familes

David L. Parnas

Alunos:

Alisson Giovanni Rosa
Fernando Peron
Dorival dos Anjos
Gustavo Cordeiro

Introdução

- Idéias iniciais sobre famílias de programas:
- *Conjunto de programas com características em comum.*

  Ex.: Versões diferentes de um programa.

- Primeira idéia de componentes.

Introdução...

- Ex.: Programas multiversão;
- Ex.: Conjunto de versões de um sistema operacional.

Motivação

- Necessidade de alterações nos requisitos dos programas;

  Ex.: Funcionalidades adicionais.

- Necessidade de alterações nas configurações de hardware;

  Ex.: Necessidade de execução em plataformas distintas.

Motivação...

- Necessidade de melhoria contínua em sistemas multiversão;

  Ex.: Otimização de códigos.

- Necessidade de existência de várias versões experimentais de um sistema;

  Ex.: Nem sempre é possível projetar-se todos algoritmos antes da implementação.

Motivação...

- Exigência anterior de equipes de manutenção e documentação específicas para cada versão;

  - O que é um caro problema para a indústria de software!

- Suposição de que custo total de desenvolvimento e manutenção será bem reduzido;

Objetivo do Trabalho de Parnas

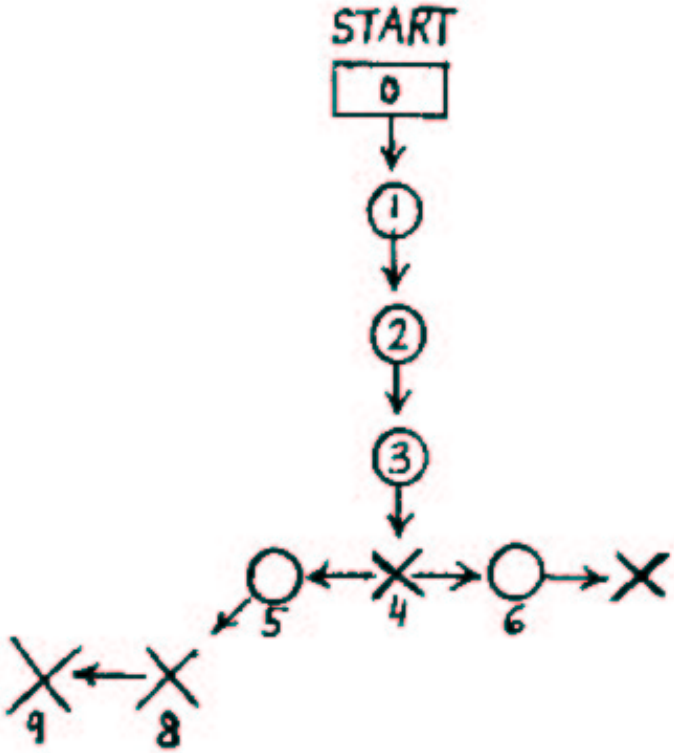
- Comparar os métodos provendo alguma idéia sobre as vantagens e desvantagens de cada um.

Método clássico

- Melhor definido como “complementação seqüencial”;

  Um membro em particular é desenvolvido até o estágio de produção. Os outros membros são desenvolvidos através da modificação deste. E assim vai...

Representação gráfica



Problemas do modelo clássico

- Decisões (por hora) inúteis herdadas de ancestrais, que geram deficiências de performance, mas que não são removidas em função da grande quantidade de reprogramação que seria necessária;

  Ex.: Ancestrais projetados para trabalhar em um ambiente diferente do atual;

Problemas do modelo clássico...

- Baixa performance em programas modificados que foram anteriormente projetados para funcionar em um diferente ambiente ou com uma carga diferente.

“Novas” técnicas

- Não é necessário o desenvolvimento de um programa completo para se obter um novo membro da família;
- Inicia-se em um estágio intermediário, ignorando-se as decisões de implementação feitas a partir deste ponto para outras versões;

“Novas” técnicas...

- Versões podem ter ancestrais comuns e não precisam ser desenvolvidas seqüencialmente;

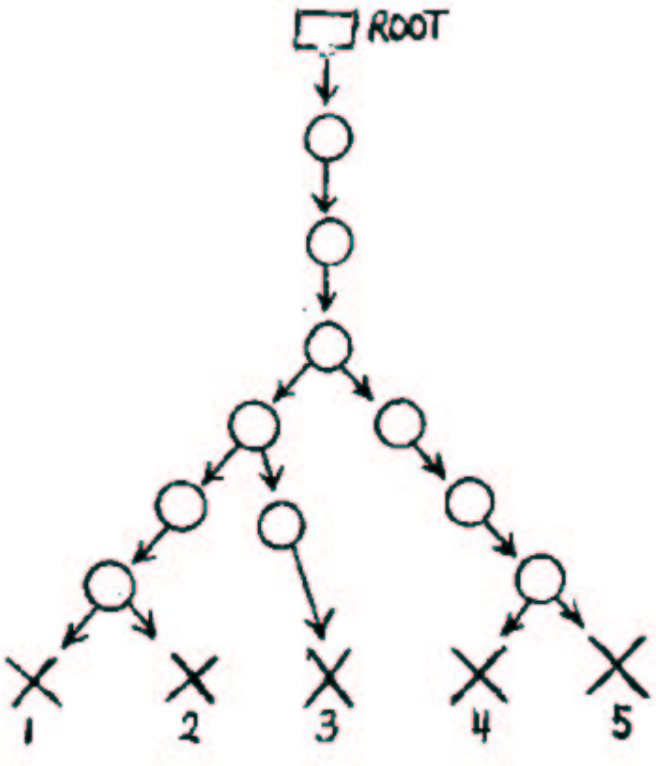
  Se o desenvolvimento de um ramo da árvore não utiliza a informação de outro, eles podem ser desenvolvidos paralelamente;
- A ordem em que as decisões são feitas possui mais significado.

  Teremos mais características comuns entre os sistemas se as decisões de projeto forem feitas o quanto antes;

“Novas” técnicas...

- A diferenciação é feita através da criação de subfamílias que irão compartilhar muitas decisões de uma família inteira;

Representação gráfica



“Novas” técnicas...

- A representação através de uma árvore é muito simplificada;

  Algumas decisões de projeto podem ser tomadas sem levar em consideração qualquer outra anterior. Assim é possível encontrar-se dois programas que fizeram uma mesma decisão de projeto mas que não é encontrada em qualquer ancestral comum.

Representando estágios intermediários

- No método clássico os estágios intermediários não foram definidos e os projetos parciais não foram bem representados;

  Como resultado a comunicação entre versões está na forma de programas completos.

- Há necessidade de métodos que descrevam precisamente programas projetados parcialmente;

  Uma representação incompleta é oferecida como contribuição ao trabalho alheio.

Dois “novos” métodos apresentados

- Refinamento gradativo;
- Especificação de módulos.

Refinamento Gradativo

- Introduzido por Dijkstra e discutido por vários colaboradores;
- Tinha como objetivo a produção de programas corretos mas uma das conseqüências foi encorajar a produção de famílias de programas.

Refinamento Gradativo ...

- Estágios intermediários representados por programas quase completos;
- Não são implementados os operandos e operadores;
- Como se os operandos e operadores fizessem parte da linguagem;
- A implementação dos operandos e operadores são postergados para estágios posteriores;

Refinamento Gradativo...

- Definição dos operadores é suficientemente abstrata para permitir uma variedade de implementações;
- As versões posteriores do programa definem uma família na qual há um membro para cada possibilidade de implementação de um operador e operando não implementado;

Refinamento Gradativo...

- Ex.: Um programa que foi escrito com a declaração do tipo de dados “pilha” e operadores “push” e “pop”. Somente nas versões posteriores é que a representação da pilha e procedimentos para executar “push” e “pop” serão introduzidos.

Programa exemplo

- Programa de números primos: Dijkstra descreveu o desenvolvimento de um programa para a impressão de números primos. O primeiro passo é o seguinte:

```
begin
  variable table p;
  fill table p with first thousand prime
  numbers;
  print table p;
end
```

Refinamento Gradativo...

- Assumiu-se um operando e dois operadores.

- Decisões postergadas:
  - Representação da tabela;
  - Método de cálculo dos números primos;
  - Formato de apresentação;

Refinamento Gradativo...

- As decisões obrigatórias:
  - Todos os números primos serão desenvolvidos antes que qualquer um seja representado.
  - Tomaremos os primeiro mil números primos.
- Os próximos membros da família serão desenvolvidos considerando diversos métodos possíveis de cálculo de números primos.

Técnica de Especificação de Módulos

- Distingue-se do método de refinamento gradativo pelo fato das representações intermediárias serem programas completos.

- São "especificações" do externamente visível comportamento coletivo de grupos de programas chamados módulos.

Técnica de Especificação de Módulos...

- Estas representações intermediárias não são escritas em uma linguagem de programação, e nunca tornam-se parte do sistema final.

- As decisões de projeto que não podem ser propriedades da família são identificados em um módulo.

Técnica de Especificação de Módulos...

- Para ocultar a representação de dados na memória, o módulo permitiria que seus usuários usassem uma função para acessar um determinado dado.

  Através do uso desse método os programas ficariam completamente independentes da representação.

Observações Comparativas

- No desenvolvimento do programa de números primos, Dijkstra é movido a fazer uma decisão prematura sobre a implementação da tabela de forma a ir mais longe.
- Todos os membros da família desenvolvida subseqüentemente deverão compartilhar aquela implementação.

Observações Comparativas...

- Caso Dijkstra decidisse voltar atrás e reconsiderar aquela decisão, teria que reconsiderar todas as decisões feitas após aquele ponto.
- O método de especificação de módulo deveria dar margem a postergação da implementação da tabela para um estágio posterior e desse modo terminar com êxito uma família ampla.

Exemplo

- Observações comparativas baseadas em um problema de sistema operacional.
- Consideremos o problema de alocação de espaço em um sistema operacional.

Exemplo

- Os programas podem diferenciar-se em pelo menos dois caminhos:

  - política;

  - implementação do mecanismo.

Exemplo

- Por política nós significamos simplesmente a regra da escolha do lugar;

- Pela implementação do mecanismo, nós significamos questões semelhantes a, como deveremos mostrar a lista dos lugares livres, quais operação nós devemos executar para adicionar um espaço livre a lista, como remover um espaço livre? etc..

Políticas

- A "melhor" política depende da natureza da demanda, i.e., a distribuição da áreas requisitadas, o quantidade de tempo desejado para que uma área será retornada, e tantas outras.

Exemplos de Políticas

- “Primeiro local" – alocar o primeiro espaço utilizável na lista;
- “Melhor local" – encontrar o menor lugar que irá encaixar;
- “Melhor local modificado" – procura por um pedaço que encaixa bem mas não deixa um pequeno fragmento inutilizável;

Exemplo

- O desenvolvimento do seguinte "programa estruturado" de semelhante algoritmo ilustra a construção de um programa abstrato que possui as propriedades de todos os quais nós estamos interessados e não prejudica nossa escolha:

Programa – Estágio 1

```
melhorEscolha := null;
WHILE NemTodosEspaçosConsiderados DO
BEGIN
  EncontrePróximoItemDaListaDeEspaçosLivre
      (candidato);
  melhorEscolha := MelhorDe
      (melhorEscolha,candidato);
END
IF MelhorEscolha = null THEN AçãoParaErros
Alocar (melhorEscolha);
Remover (melhorEscolha);
```

Decisões do Projeto no Estágio 1

- Ninguém está autorizado a adicionar espaços livres enquanto se procura por um espaço livre.
- Não podemos construir um programa em que duas buscas possam ser realizadas simultaneamente.
- Os espaços candidatos não são excluídos da lista de espaço livre enquanto ele está sendo considerado.
- Nenhum teste da possível alocação é feita antes de buscar na lista.

Estágio 2

- Lista de espaços livre agora é definida por 1 array bi-dimensional.
  - Cada linha representa um pedaço de espaço de memória livre;
  - Linha 1 conterá o primeiro espaço livre de memória;
  - Linha N conterá o último espaço livre de memória;
  - Entre 1 e N, estarão representados todos os espaços de memória livre válidos sem definição de ordem ou informação contida.

Programa Estágio 2

```
melhorEscolha := 0;
candidato := 0;
WHILE candidato <> N DO
  BEGIN
    candidato := candidato + 1;
    melhorEscolha :=
                 MelhorDe(melhorEscolha,candidato);
  END;
IF melhorEscolha = 0 THEN AçãoDeErro;
Alocar (melhorEscolha);
Remover (melhorEscolha);
```

Considerações do Estágio 2

- Alteração tipos das variaveis “melhorEscolhã” e “candidato” p/ implementar “NemTodosEspaçosConsiderados”;
- As suposições não permitem alterações nas linhas da tabela e nem implementar a politica de decisão “MelhorDe”;
- Não podemos implementar “Remover” sem saber se todo espaço será alocado;

Programa – Estágio 3

- Um membro concreto familia;
- Decisão tomadas:
  - Iremos encontrar o menor espaço que sirva para alocação do programa;
  - Uma vez escolhido este espaço, todo ele será alocado;

```
melhorEscolha := 0;
candidato := 0;
antigoT := 99999999;
WHILE candidato <> N DO
  BEGIN
    candidato := candidato + 1;
    T:=(Final(candidato)-Inicio(candidato));
    IF (T>=requisitado) AND (T<antigoT) THEN BEGIN
      melhorEscolha := candidato;
      antigoT := T;
    END;
  END;
IF melhorEscolha = 0 THEN AçãoDeErro;
Alocar (melhorEscolha);
N := N - 1;
FOR I := melhorEscolha UNTIL N DO BEGIN
  End [I] := End [I + 1];
  Start [I] := Start [I + 1];
END;
```

E se...

- Não quisessemos alocar o menor espaço disponível, mas apenas o espaço que o programa esta precissando?

Ou

- Nós representássemos os espaços disponíveis, pelo endereço inicial e o tamanho disponível a partir dele?

Primeira Situação Possível

- Modo clássico:
  - Alterar o programa a partir do último estágio sem retornar aos estágios intermediários;

  Neste caso:
  - Temos que modificar o programa mostrado no estágio 3;
  - Requer estudo cuidadoso do programa;
  - Programador muito familiarizado com o código;

Segunda Situação Possível

- Nós usarmos o desenvolvimento de programas estruturados.

  Neste caso:

  – Opção de retornar ao programa do estagio 2;
  – Todas as suposições feitas nele continuam valendo;
  – Complementar até produzir um programa concreto;
  – Pode ser feito por qualquer um.

Como as Especificações dos Módulos Definem uma Família

1. Métodos de implementação usados nos módulos;
2. Variações nos parâmetros externos;
3. Uso de subconjuntos;

Qual Método Utilizar

- Os dois métodos não são nem equivalentes nem contraditórios. São complementares.
- Ambos são baseados na mesma idéia:
  1. representação precisa do estágio intermediário em um desenho de programa; e
  2. Postergação de certas decisões, enquanto continua fazendo progresso em direção ao programa completo.

Conclusão

- Uma das dificuldades de aplicar os **recentes** conceitos programação estruturada é que não há critérios de como se deva avaliar a estrutura de um sistema em uma base objetiva.
- Os aspirantes da profissão devem ir aos "mestres" e pedir uma avaliação. O "mestre" poderá dizer o que é bom ou não para o sistema.

Conclusão...

- O conceito de programação em famílias nos dá uma maneira de analisar a estrutura do programa mais objetivamente.

Conclusão...

- Pode-se considerar o desenvolvimento do programa como bom, se as decisões anteriores tiverem excluído programas que não interessavam, que eram indesejados ou eram desnecessários. As decisões que removem programas desejáveis deveriam ser também postergadas para um determinado estagio ou confinadas em um subconjunto de código bem definido.

Referências

- Baseado no artigo de David L. Parnas publicado em IEEE Transactions on Software Engineering Vol. SE-2, nº 1,Março de 1976.